The PREDA Language

A Distributed Programming Language for General Smart Contracts on Sharded Blockchains and Cross-Chain Bridges

PREDA DEV TEAM

PREDA, (**P**arallel **R**elay-and-**E**xecution **D**istributed **A**rchitecture), is a novel programming model for general smart contracts dealing with multi-chain blockchain systems, in which logic execution and ledger states are partitioned and distributed across chains. In this document, we introduce the core features of the PREDA programming language with code examples.

The PREDA language is a high-level programming language with curly-bracket and Algol-like syntax style for general smart contract development. It is naturally parallelized and distributed based on the proposed programming model, which provides object-oriented scope syntax and lambda function syntax to support the two core features of the PREDA model. The PREDA language aims to maximize the interoperability with sequential, non-distributed programming languages (e.g. Solidity), which allows cross-language invocation with unifies type system and asynchronous calls to/from languages with sequential programming model.

Compilation system and execution engine of the PREDA language is implemented as compiling to intermediate representation, and then building to platform-specific native binaries or virtual machine bytecodes. Solidity compiler and Ethereum Virtual Machine are integrated as parts of the system to realize the interoperability and cross-language compilation. Compilation system generates function symbols and unified argument serialization, which enable cross-language invocation and relay transaction composition based on functional relay semantics.

1 SCOPES AND RELAYS

The PREDA programming language provides syntax to decompose states of a smart contract into **scopes** with different synchronization requirements within- and across-chains/shards, which enables state sharding for general smart contracts rather than payment contracts. The PREDA language also provides syntax to decompose execution flow within a contract function into multiple **microfunctions** that are relayed and executed in different shards based on their data dependencies of states in different scopes.

1.1 Built-in Scopes

PREDA has three built-in scopes, i.e., **global**, **shard** and **address**. They correspond to the data partitioning structures in sharded blockchains. Programmers specify the scopes of the contract variables, and the underlying blockchain system initiates and executes contract functions on the scopes accordingly:

- **Global scope** is used to define common contract states that must be shared and synchronized by all shards. A global scope state is effectively a singleton that is available throughout the network. In addition, all deployed contracts and their codes belong to the global scope. A contract can import and interpret any function from another deployed contract.
- **Shard scope** is used to define common contract states within a shard, but not specific to a user. A shard scope state must be instantiated and updated independently in each shard.
- **Address scope** is used to define contract states for each user. An address scope state is initiated and updated individually in each address. This is the finest granularity a contract can expose to the underlying system.

Author's address: PREDA Dev Team, devteam@preda-lang.org.

From the perspective of object allocation, the global scope is the equivalent of a conventional smart contract. Everything defined in the global scope has an instance on all blockchain nodes and the instances are consistent globally. The shard scope defines a state that has one instance on each blockchain node of the shard and the instances are consistent in the shard. The address scope defines a state that has one instance for an address and only blockchain nodes belonging to the same shard that have the address have the instances.

The keywords **"@global"**, **"@shard"**, and **"@address"** are used before variables and functions to define their scopes as below. Note that when a state variable or function is defined without specifying a scope, it defaults to **"@global"**.

```
contract MyContract {
    @global  uint32  numTotalAccounts;                    // only one instance globally
    @shard   uint32  numAccountsInShard;                  // one instance per shard
    @address uint512 addressBalance;                      // one instance per address
}
```

Listing 1. Three built-in scopes in PREDA

1.2 User-defined Scopes

In addition to the built-in scopes, programmers can define their own scopes as **"@scopeName = global by typeName;"** . It means that there is a user-defined scope named "scopeName", which is indexed by "typeName" on chain. The "typeName" can be one of the following by-default datatypes in PREDA: uint16, uint32, uint64, uint128, uint256, uint512, and address. Note that even when two scopes are defined with the same "typeName", they are regarded as different scopes. A user-defined scope can be used after it's defined in a PREDA program as below:

```
contract C {
  @myScope = global by uint32;
  @myScope string myStr;
}
```

Listing 2. User-defined scopes in PREDA

In this example, the scope **"myScope"** can have up to $2^{32} - 1$ instances that are indexable by a uint32 value. Which shard each of these instances resides in is decided by the underlying blockchain system and transparent to the contract.

1.3 Relay

When a contract function executing in a scope needs to access data defined in another scope that the function cannot directly access, a relay is required in PREDA. This means the transaction execution context is switched from one scope to another but the execution logic is continued. Listing 3 shows the PREDA implementation of the token transfer contract, which is equivalent to the Solidity function `transfer`. The keyword **"@address"** is used to define the scope of contract variables and functions that belong to an address. The `transfer` function executing at the scope of the sender address does the withdraw on the balance of the sender and then uses a relay to call the `deposit` function, as the deposit is executed at the scope of the recipient address.

```
contract Token {
    @address bigint balance;
    @address function bool transfer(address to, bigint amount) export
    {
        if(balance >= amount)
        {
            balance -= amount;
            relay@to deposit(amount);
```

```
9               return true;
10          }
11          return false;
12      }
13
14      @address function bool deposit(bigint amount)
15      {
16          if (amount <= 0) return false;
17          balance += amount;
18          return true;
19      }
20  }
```

Listing 3. Corresponding code snippet of `transfer` function in PREDA for ERC20 transfer in Solidity

As shown in this example, **"relay"** is the keyword to specify a relay statement and **"@"** is another keyword to specify the destination of a relay. A formal representation of a relay is:

$$relay@address \quad function(parameters) \tag{1}$$

Comparing the code example in Listing 3 with the Solidity implementation of ERC20 token transfer contract [3], one can find that the PREDA implementation converts the map `balances` in the Solidity implementation to an integer variable `balance` and does not specify whose `balance` is as Solidity did, e.g., message sender's balance as `balances[msg.sender]`. This is because the global variable `balances` defined in Solidity is converted to an address variable `balance` in PREDA. The execution context is switched from the sender's address scope to the recipient address scope with the relay statement.

Note that although PREDA doesn't specify how to implement a relay in the underlying blockchain system, a relay is recommended to be implemented as an asynchronous relay transaction. That is because (1) transaction is the finest granularity of communication between blockchain nodes and almost all core modules of a blockchain, e.g., transaction submission and propagation, block generation and verification, security mechanisms, etc., are built upon transactions; and (2) in a blockchain, it is unknown when a transaction will be executed. In the worst case, a transaction is discarded by blockchain nodes after an extremely long wait. The semantics of a relay must be asynchronous.

Figure 1 shows how the PREDA model executes transactions in parallel. Assuming there are four shards A, B, C, D, and two user-initialized transactions φ and φ', which can be the transactions invoking the PREDA function `transfer` shown in Listing 3. The withdraw operation ρ that only involves the state in shard A is handled by a miner in shard A. If the account balance satisfies the cost of this withdraw operation, a block $i+1$ carrying the transaction φ is created by the miner and appended to the chain of shard A. After that, a relay transaction carrying the deposit operation ϕ is composed in shard A and forwarded to shard B. The deposit operation ϕ that only involves the state in shard B can be executed by shard B. Once the relay transaction is picked up by another miner in shard B, operation ϕ is executed, concluding the complete of the transaction φ. Similarly, the withdraw operation ϕ of transaction φ' is executed in shard C and its deposit operation ρ is executed in shard D. The withdraw operations on addresses in different shards can be executed in parallel and similarly to the deposit operations.

1.4 Relays between Shard and Global Scopes

In the PREDA token transfer example (shown in Listing 3), the relays occur between address scopes. In this subsection, we shown an example that the relays occur between the shard and global scopes. Listing 4 shows the PREDA Ballot contract. It is functionally equivalent to the Solidity Ballot contract, but is designed to execute a large number of voting transactions in parallel on a sharded blockchain. The PREDA implementation is quite complex, but shows many features, e.g., relay transactions across different types of scopes. The main ideas of the

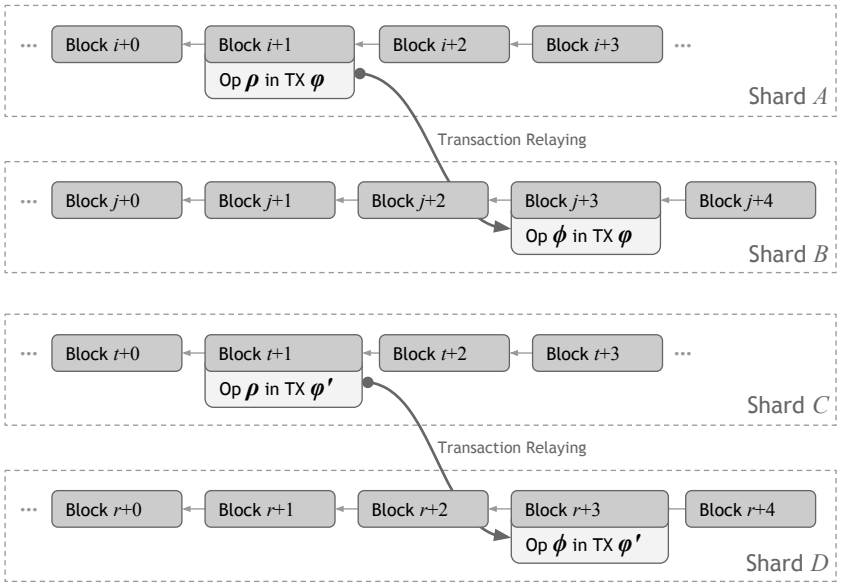

Fig. 1. Parallel execution of transactions φ and φ' in different shards.

PREDA Ballot contract can be explained as below: first, the contract allows the voters to vote in different shards based on address-to-shard mapping (by the underlying blockchain); and second, the contract accumulates the voting results of each shard to the final voting result. The PREDA Ballot contract can avoid reading and writing the shared contract state (i.e., the final voting result) simultaneously throughout the network, when processing voting transactions issued by the voters. In this case, we show how voting is implemented in the shard scope and how the final voting result is accumulated in the global scope.

```
contract Ballot {
    struct Proposal {
        string name;
        uint64 totalVotedWeight;
    }

    struct BallotResult {
        string topVoted;
        uint32 case;
    }

    @global address controller;
    @global uint32 current_case;
    @global array<Proposal> proposals;
    @global BallotResult last_result;

    @global uint32 shardGatherRatio;
    @global function shardGather_reset(){ shardGatherRatio = 0u; }
    @global function bool shardGather_isCompleted(){ return shardGatherRatio == 0x80000000u; }
    @global function bool shardGather_gather()
    {   shardGatherRatio += 0x80000000u>>__block.get_shard_order();
        return shardGatherRatio == 0x80000000u;
    }

```

```
25      @shard array<uint64> votedWeights;
26      @address uint64 weight;
27      @address uint32 voted_case;
28
29      @address function bool is_voting()
30      {
31          return last_result.case < current_case;
32      }
33
34      @address function init(array<string> names) export {
35          relay@global (^names){
36              for (uint32 i = 0u; i < names.length(); i++) {
37                  Proposal proposal;
38                  proposal.name = names[i];
39                  proposal.totalVotedWeight = 0u64;
40                  proposals.push(proposal);
41              }
42              current_case++;
43              last_result.case = 0u;
44              last_result.topVoted = " ";
45          }
46      }
47
48      @address function bool vote(uint32 proposal_index, uint32 case_num) export {
49          if(case_num == current_case && case_num > voted_case && proposal_index<proposals.length())
50          {
51              voted_case = case_num;
52              votedWeights.set_length(proposals.length());
53              votedWeights[proposal_index] += weight;
54              return true;
55          }
56          return false;
57      }
58
59      @address function finalize() export {
60          relay@global (){
61              shardGather_reset();
62              relay@shards (){
63                  __debug.print("Shard Vote: ", votedWeights);
64                  relay@global(auto shardVotes = votedWeights) {
65                      for(uint32 i=0u; i<shardVotes.length(); i++)
66                          proposals[i].totalVotedWeight += uint64(shardVotes[i]);
67
68                      if(shardGather_gather()) {
69                          last_result.case = current_case;
70                          uint64 w = 0u64;
71                          for(uint32 i=0u; i<proposals.length(); i++) {
72                              if(proposals[i].totalVotedWeight > w)
73                              {
74                                  last_result.topVoted = proposals[i].name;
75                                  w = proposals[i].totalVotedWeight;
76                              }
77                          }
78                      }
79                  }
80              }
81          }
82      }
83  }
```

Listing 4. The PREDA ballot contract

In the address scope function `init`, the controller of the contract is responsible for initiating the global scope variable `proposals`. This is done by issuing a relay transaction from the address scope to the global scope, which is corresponding to broadcast a relay transaction to the network in the underlying blockchain and to initialize the `proposals` and `last_result` variables. After the execution of the `init` function, the global scope variable `proposals` is initialized on all blockchain nodes.

In the address scope function `vote`, each voter can **directly** read the global scope variable `proposals` and also **directly** write the shard scope variable `voteWeights`, because the global scope variable is globally consistent, and the shard scope variable is consistent in each shard. At runtime, user-initiated transactions calling the `vote` function are executed by different shards based on address-to-shard mapping. Each transaction only changes the shard scope variable `voteWeights` in the corresponding shard.

In the address scope function `finalize`, the controller of the contract issues a relay transaction from the address scope to the global scope to reset the global scope variable `shardGatherRatio` and request all shards to report the value of the shard scope variable `voteWeights`. Each shard then sends a relay transaction with the value of `voteWeights` to the global. At the global scope on each blockchain node, the partial voting results `voteWeights` are accumulated to the final voting result `proposals`. After receiving the partial voting results from all shards, i.e., in the conditional sentence of calling the `shardGather_gather` function, the winner of the voting proposals is calculated and set as `last_result`.

2 MORE LANGUAGE FEATURES

PREDA language has enriched features. We introduce several of them in this section. Please refer to PREDA Language Specification [4] for more details.

2.1 Relay Statement with Lambda Expression

As shown in the previous example of the PREDA ballot contract, a relay statement can be defined as a lambda function. The format is quite similar to defining a function, except that:

- A function name is not needed. The compiler automatically generates a name for it.
- The scope of an anonymous function can be address (user-defined), shard, or global, based on the relay type.
- For each parameter, an argument must be provided as well.
- It is possible to use the "auto" keyword as the parameter type. In this case, the type is taken from the corresponding argument expression.

The relay function body is executed on the per-address context of target address, the per-shard context of the target shards, or the global context. It cannot be mixed with the current context scope on which the relay statement is invoked. There's two ways to specify a parameter in the relay lambda.

```
1 relay@someAddress (..., ^identifier, ...){
2 }
3 relay@someAddress (..., auto identifier = identifier, ...){
4 }
```

Listing 5. Relay Statement with Lambda Expression

They are equivalent but the first one is a simplified expression, where the ^ operator is used to capture variable by value, programmers can thus pass parameters to the function without renaming them.

2.2 System-reserved Functions

System-reserved functions are a group of special functions with the names reserved by PREDA for special purposes. They don't always have to be defined by a contract. But when they are, the definition must match a

certain signature and will be invoked by the system at certain points. They are const functions and may emit relay transactions to do the actual task if needed. In addition, system-reserved functions may not access the transaction context (because they are not invoked by a transaction), or any part of the block context that is of payload or mined dependency (because they are executed before transactions in a block). There are two system-reserved functions in PREDA .

- **on_deploy():** it is a global function that is invoked when a contract is deployed. It can be used to do some initialization of the contract state. Its signature is: `function on_deploy()`.
- **on_scaleout():** it is a shard function that is invoked when the scaleout of the sharded blockchain happens, i.e. when the shard order of the blockchain system is increased by 1, i.e., from $n-1$ to n and the total number of shards doubles from 2^{n-1} to 2^n. On scaleout, each of the old 2^{n-1} shards is forked to two new shards: from $shard[i]$ to $shard[i]$ and $shard[i+2^{n-1}]$, where $0 <= i < 2^{n-1}$. The on_scaleout function is called 2^n times, once per shard. It can be used to split the old per-shard contract state into the two new shards. The signature is: `function on_scaleout(bool)`. The boolean parameter tells whether the current shard is forked in place (when false), or with offset 2^{n-1} (when true).

Note that although the system-reserved functions are defined in the PREDA language specification, they are not mandatory for underlying sharded blockchain systems. If a sharded blockchain doesn't support a system-reserved function, a PREDA contract using the function can still be compiled and executed on the sharded blockchain but doesn't have the functionality defined by the system-reserved function.

2.3 Multiple Contracts

In PREDA , a contract could interact with other contracts that are already deployed on the chain. To interact with another contract that contract must first be imported to the current contract as below:

```
import DAppName.ContractName [as AliasName];
```

`DAppName` and `ContractName` are the corresponding names assigned when deploying the imported contract. `AliasName` is an optional arbitrary identifier to reference it in the current contract. If `AliasName` is not given, `ContractName` will be used instead for referencing. The import must be declared before the new contract definition.

The PREDA language specification supports explicit import and implicit import. When a contract is imported by an import directive, it is explicitly imported. Besides that, a contract could also be implicitly imported if it is indirectly imported, like in the following example:

```
contract ContractA{
}

import MyDApp.ContractA as A;                          // ContractA is explicitly imported
contract ContractB{
}

import MyDApp.ContractB as B;                          // ContractA is implicitly imported
contract ContractC{
}
```

An implicitly-imported contract doesn't have a user-defined alias and can be referenced by its contract name by the compiler. In the above example, `MyDApp.ContractA` is referenced as `ContractA` in Contract C. To have a specific alias, it could be explicitly imported again. For example:

```
import MyDApp.ContractB as B;                          // ContractB is explicitly imported
import MyDApp.ContractA as A;                          // ContractA is explicitly imported as A
contract ContractC{
}
```

After importing a contract, all user-defined types and scopes from it could be accessed under the contract alias. As shown in the example below, after `ContractB` imports `ContractA`, the function `f` in `ContractB` has the scope `myScope` defined in `ContractA`.

```
contract ContractA{
  struct S{
    int32 i;
  }
  enum E{
    E0,
    E1
  }
  @myScope = global by address;
}
import MyDApp.ContractA as A;
contract ContractB{
  @A.myScope {
    A.S s;
    A.E e;
    function f(){
      s.i = 1i32;
      e = A.E.E0;
    }
  }
}
```

Similar to user-defined types, public functions defined in other contracts could also be directly referenced via the alias.

```
contract ContractA{
  struct S{
    int32 i;
  }
  enum E{
    E0,
    E1
  }
  @myScope = global by address;
  @myScope {
    function f(S s, E e) public{                          // only public functions can be called by other contracts
    }
  }
}
import MyDApp.ContractA as A;
contract ContractB{
  @A.myScope {
    A.S s;
    A.E e;
    function f(){
      A.f(s, e);                                          // call the public function f() from MyDApp.ContractA
    }
  }
}
```

The basic scope visibility rules hold for cross-contract calls, i.e., each scope can only call functions in the same scope, in the shard scope and const functions in the global scope.

2.4 Interfaces

Interfaces provide another way to work with multiple contracts. While only known contracts can be imported, interfaces enables interaction with arbitrary contracts that implements it. Interfaces are defined at the contract level. Each interface is a set of function definitions with empty bodies. Similar to regular functions, the functions of an interface must also reside in scopes:

```
contract A {
  interface Addable {
    @address {
      function Add(uint64 value);
    }
    @global {
      function uint64 GetTotal() const;
    }
  }
}
```

The above contract defines an interface `Addable` with two functions, each in a different scope. Interfaces can use scopes freely like scopes in contracts, including user-defined scopes and imported scopes from other contracts.

Contracts can choose to implement interfaces using the **implements** keyword at definition. A contract can choose to implement arbitrary number of interfaces, which can either be those defined in the same contract, or imported interfaces from other contracts.

```
import A;
contract B implements A.Addable, Printable {                // use "implements" to implement interfaces
  interface Printable {
    @global {
        function Print() const;
    }
  }

  @global {
     uint64 total;
     function uint64 GetTotal() public const {              // GetTotal() for A.Addable
       return total;
     }
     function Print() public const {                        // Print() for Printable
       __debug.print(globalTotal);
     }
  }
  @address {
     function Add(uint64 value) public {                    // Add() for A.Addable
       relay@global (^value) {
         total += value;
       }
     }
  }
}
```

The above contract implements two interface: `Printable` defined in the contract itself, and `Addable` defined in contract `A` from the previous section.

To implement an interface, a contract must implement all the functions defined in that interface, and the signature of the implemented function must match exactly the definition in the interface, i.e. the same function name, parameter list and type, return type, const-ness and scope. In addition, interface function must be implemented as **public**, since they are used for cross-contract calls. When a contract implements an interface, other contracts can interact with it via the interface. For example:

```
import B;                                                    // A is implicitly imported via B
contract C {
  @address {
    function test() {
      A.Addable addable = A.Addable(B.__id());               // initialize addable with contract B's id
      addable.Add(100u64);                                   // Calls B.Add() via the interface
    }
  }
}
```

In this example, a variable of interface type `A.Addable` is defined. Interface types can be initialized with a contract id. Here, it is initialized with a B's id using the build-in function `__id()` that is automatically generated for each contract. Once an interface variable is initialized, it can be used to call any function defined in the interface and routed to the corresponding implementation in contract `B`. With interfaces, a contract can interact with any other contract that implements the interface without knowing them. For example:

```
import A;
contract Adder {
  @address {
    function Add(A.Addable addable, uint64 value) public {
      addable.Add(value);
    }
  }
}
```

When the interface is defined, there is no need to import any other contract other than `A`. The function `Add` accepts an `A.Addable` interface as parameter, which could possibly be initialized by the id of any other contract that implements `A.Addable`. Note that if calling a function on an interface variable that is uninitialized, or initialized with the id of a contract that actually doesn't implement the interface, an error would occur and the contract execution will fail immediately.

3 DEPLOYMENT AND EXECUTION ENGINES

We have implemented a preview toolchain for the PREDA programming language. The PREDA language preview toolchain includes the pre-built PREDA compilers and execution engines, VSCode extensions, and sample smart contracts and execution scripts. Programmers can use VSCode to edit, compile, and execute PREDA smart contracts in a local blockchain simulator. The chain simulator uses multiple threads to mimic multiple shards of a sharded blockchain (one for each) on a single node. Programmers can also use the VSCode extension to check the contract execution results, e.g., address states on the chain. The supported command lines of the chain simulator and the script syntax of the chain simulator can be found in the PREDA toolchain user manual [5].

3.1 Contract State Management in Chain Simulator

In the implementation of the PREDA chain simulator, we use the Map of the contract ID to the contract states as the basic data structure. The basic chain state is divided into pieces represented as `<scope, [shard/address], contract map>`. The chain simulator uses this structure to track partial changes to the entire chain state. At runtime, the execution of transactions, including those in chain blocks and orphaned blocks, results in newly modified contract states. We store incremental changes to contract states as follows:

```
<Global State>   ::= <Contract Map>
<Shard State>    ::= <Contract Map>
<Address State>  ::= MAP(<Address> => <Contract Map>)
<Contract Map>   ::= MAP(<Contract Id> => <State History>)
<State History>  ::= ARRAY(<State on Chain>)
```

`<State on Chain> ::= <Height, Block Id, Opaque State Data>`

Note that the array `<State History>` is sorted as the chain height increases. In the chain simulator, the total chain states include: (1) the base chain states for archived blocks that are not reverted, and (2) a history of incremental changes made by all blocks since the height of the last archived block, including blocks in forked chains.

3.2 Contract Compilation

We employ a two-stage process to compile smart contracts written in PREDA to native code. The first stage is a transpiler that interprets the PREDA source code and converts it to an intermediate representation, which is then consumed by the second stage to generate the binary code. We use C++ source code as our intermediate representation.

Our transpiler uses ANTLR [1] to generate the parser code based on the PREDA language specification. Then, the transpiler walks through the abstract syntax tree (AST) to extract state variables, user-types, and function definitions, and outputs the corresponding C++ code. The output of the transpiler for each contract is a single C++ complication unit. To compile C++ code to binary code, we use MinGW-w64 [2] on Windows and g++ on Linux. Their output are dynamic libraries that can be loaded on demand at runtime when a contract needs to be executed on the chain simulator.

Other than the local chain simulator, we are developing an experimental sharded blockchain for PREDA , with the supports of blockchain wallets and explorer services. We are also integrating EVMone into the chain simulator to implement the interoperability between Solidity and PREDA.

REFERENCES

[1] [n. d.]. ANother Tool for Language Recognition. https://www.antlr.org/.
[2] [n. d.]. MinGW-w64. https://www.mingw-w64.org/.
[3] EthereumDev. 2020. Transfers and Approval of ERC-20 Tokens from a Solidity Smart Contract. https://ethereum.org/pt-br/developers/tutorials/transfers-and-approval-of-erc-20-tokens-/\from-a-solidity-smart-contract/.
[4] PREDA Dev Team. 2022. PREDA Language Specification.
[5] PREDA Dev Team. 2022. PREDA Toolchain User Manual.